Technische Universität Braunschweig

This is an author produced version of :

Hardware and Software Task Scheduling for ARM-FPGA Platforms

Article:

A. Dorflinger et al., "Hardware and Software Task Scheduling for ARM-FPGA Platforms," 2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2018, pp. 66-73.

DOI: 10.1109/AHS.2018.8541481

©2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, including reprinting/republishing this material for advertising or promotional purposes, collecting new collected works for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

https://doi.org/10.24355/dbbs.084-202105041433-0

Hardware and Software Task Scheduling for ARM-FPGA Platforms

Alexander Dörflinger, Mark Albers, Johannes Schlatow, Björn Fiethe, Harald Michalik
Institute of Computer and Network Engineering (IDA)
TU Braunschweig
Braunschweig, Germany
{doerflinger, albers, schlatow, fiethe, michalik}@ida.ing.tu-bs.de

Phillip Keldenich, Sándor P. Fekete
Algorithms Group
TU Braunschweig
Braunschweig, Germany
{p.keldenich, s.fekete}@tu-bs.de

***Abstract*—ARM-FPGA coupled platforms allow accelerating the computation of specific algorithms by executing them in the FPGA fabric. Several computation steps of our case study for a stereo vision application have been accelerated by hardware implementations. Dynamic Partial Reconfiguration places these hardware tasks in the programmable logic at appropriate times. For an efficient scheduling, it needs to be decided when and where to execute a task. Although there already exist hardware/software scheduling strategies and algorithms, none exploit all possible optimization techniques: re-use, prefetching, parallelization, and pipelining of hardware tasks. The scheduling algorithm proposed in this paper takes this into account and optimizes for the objectives latency/throughput and power/energy.**

I. Introduction

Tightly coupled ARM-FPGA systems, such as the Xilinx Zynq-7000 SoC, benefit from high bandwidth interfaces between hard-wired processor cores and the FPGA fabric. This feature allows efficient acceleration of data-intensive computations by parallelizing the execution in the programmable logic. If the processing power of the hard-wired ARM-based processor system does not suffice for an application, the computation of large data payloads can be outsourced and executed in the programmable logic as a hardware accelerated task. The number of concurrently dispatchable tasks is limited by the available resources in the programmable logic. However, the ability of SRAM-based FPGAs to support Dynamic Partial Reconfiguration (DPR) allows a very flexible use of the available resources in a Time-Space Partitioning (TSP) manner. If the given resources do not suffice for a static and concurrent implementation of all required hardware tasks, DPR allows to sequentially load and execute those tasks.

The continuous enhancement of DPR support in the FPGA vendors' toolchains helps to propagate the use of DPR in embedded systems. Also for safety-critical/mixed-criticality systems, DPR promises an increase of computation performance and a reduction of power consumption. For such systems, security and real-time aspects need to be considered. By developing a framework for hardware acceleration through DPR for the highly secure Genode OS [1], security aspects are already addressed [2]. However, real-time aspects have not been covered adequately yet.

Therefore we focus in this paper on hardware/software task scheduling for ARM-FPGA platforms that satisfies real-time requirements and guarantees latency or throughput rates. Additionally, the developed scheduling algorithm optimizes for low power and low-energy consumption. This makes the approach interesting for applications with tight constraints on power/energy, which are e.g. inherent to space missions.

The rest of this paper is organized as follows. Section II describes a platform suited for efficient hardware task executions. The exemplary stereo vision application introduced in Section III makes use of this platform. Section IV discusses constraints and objectives of a scheduling algorithm, which is explained in Section V. The developed algorithm is evaluated in Section VI.

II. Platform for Hardware Task Acceleration

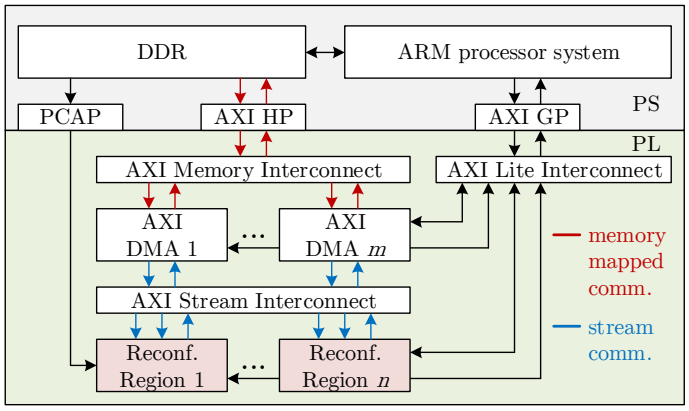

Fig. 1. Platform for hardware accelerated task execution in a Zynq-7000 SoC.

Fig. 1 depicts a flexible platform for hardware task acceleration implemented on a Xilinx Zynq-7000 SoC with a processing system (PS) and programmable logic (PL). The platform provides n reconfigurable regions in which hardware modules for accelerated task execution can be placed. Before execution, a region has to be configured via the Processor Configuration Access Port (PCAP) with the partial bitstream of the hardware module. One AXI High Performance (AXI HP) interface is used for exchanging processing data between

https://doi.org/10.24355/dbbs.084-202105041433-0

the PS and PL with a maximum theoretical bandwidth of up to $2400\,\mathrm{MByte/s}$ in both read and write direction [3]. Parametrization data for hardware tasks is communicated via the AXI General Purpose (AXI GP) interface. The AXI Stream Interconnect allows reconfigurable regions to exchange processing data with each other, or to read from and write to DDR memory via m DMA IP-cores, which translate between memory mapped and stream communication. Related work makes use of similar platforms for hardware acceleration by DPR. [4], [5], [6], and [7] use different concepts for transferring processing data, but the overall architecture is comparable.

The interface of a reconfigurable region needs to be selected carefully. On the one hand, the interface should be kept as simple as possible. All interface signals cross the border between the static and reconfigurable FPGA parts, which restricts their routing and requires decoupling mechanisms. On the other hand, the interface should be as flexible as possible. Different hardware tasks have different input and output data amounts and types, which has to be mapped to the generic region interface.

As depicted in Fig. 1, a reconfigurable region consumes up to two input streams and generates a single output stream in the proposed platform. This satisfies the communication needs of a wide range of coarse-grained signal/image processing operations. Additionally, arbitrary operation parameters (thresholding values, etc.) can be set via the AXI Lite Interface. However, the defined interface is not suited for some applications, e.g., a QAM demodulation would need two outputs. Such algorithms require an adapted platform, interleaving of stream data, or a higher decomposition level. A very fine-grained decomposition ultimately results in separate arithmetic operations consisting of two operands and one result, which also fits to the selected interface definition.

The size of reconfigurable regions is another important decision to be taken at design time. A varying size allows a flexible use of FPGA fabric resources. Small regions are created for efficient execution of simple algorithms, large regions with a higher number of logic-, DSP- and RAM resources allow the implementation of more complex algorithms.

Due to the routing and resource overhead in the FPGA fabric required by each reconfigurable region interface, a large number of regions will drop the overall resource usage efficiency. Hence, it is beneficial to keep the decomposition level low and to create larger hardware modules and regions that are able to execute several computation steps in conjunction.

For an efficient scheduling, several optimization techniques for hardware task acceleration have been identified. Four out of five can be applied with the proposed platform and can be exploited by a scheduling algorithm.

1) *Re-use* of hardware modules avoids lengthy reconfiguration processes, because a module is already configured. This helps reducing the overall execution time and keeps the memory bandwidth impact of configuration data transfers low. However, a reconfigurable region cannot execute another task while waiting for re-use, possibly resulting in long stall times.

2) *Prefetching* hides reconfiguration time by reconfiguring a region with the proper hardware module before activation of a task. The task can execute immediately after activation, under the assumption of a correct prediction of upcoming tasks.

3) *Parallelization* can be applied for tasks that operate on their processing data in a streaming fashion. The AXI Stream Interconnect allows not only for processing data transfers from and to DDR memory, but also between reconfigurable regions. If a preceding hardware task produces data that is consumed and processed in a succeeding hardware task, the data stream can be forwarded directly. The succeeding task processes the first datum as soon as it has been computed in the preceding task, which occurs after a short latency (see Fig. 2). For operations on large datasets, this latency can be neglected compared to a task's execution time. As both tasks are executed almost in parallel, the total makespan (time elapsing from task graph activation to termination of last node) of a task graph can be reduced. Furthermore, write back to and read from DDR memory is avoided, which reduces DMA transactions. Applying this technique avoids stalls caused by waiting for a free DMA channel. As the AXI Stream Interconnect has very limited data buffering resources, parallel tasks have to be synchronized and may not start their computation before all hardware modules of parallel tasks are configured in the FPGA fabric, resulting in potential stalls. This synchronization also thwarts fast tasks, which inherit the execution time of the slowest task executed in parallel.

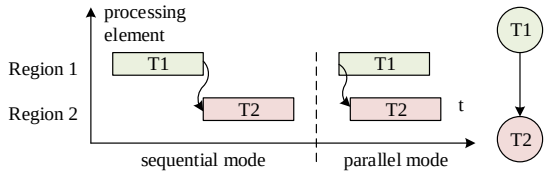

Fig. 2. Tasks executed in sequential and parallel mode.

4) *Pipelining* of consecutive iterations of a task graph can increase the repetition rate and hence throughput. For throughput considerations, it is not sufficient to only calculate the makespan of a task graph. Additionally, the resulting task graph schedule needs to be examined for potential overlapping executions in different reconfigurable regions. E.g., in Fig. 3 the execution of task T4 from the first task graph iteration overlaps with a reconfiguration process of the next task graph iteration.

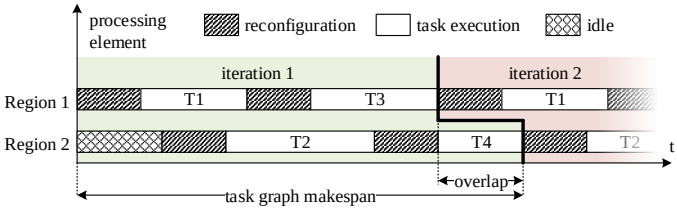

Fig. 3. Pipelining two task graph executions.

https://doi.org/10.24355/dbbs.084-202105041433-0

5) *Relocation* of hardware tasks is the process of moving an already scheduled task from one reconfigurable region to another and is discussed for example in [6]. This process requires a hardware task to be interrupted and resumed at a later point in time, or in other words, preemption of hardware tasks. Preemption requires capturing and restoring the hardware task's state, which has been discussed in some related work [8]. However, this process adds complexity to the design and is not completely fail safe from our point of view, e.g., interrupting a DMA transfer could result in unexpected behavior. Besides, we do not expect a significant reduction of makespan or throughput gain as additional reconfiguration processes prolong the overall solution. Hence, relocation of hardware tasks and preemption will not be supported by our platform.

III. Stereo Vision Case Study

Imaging algorithms are generally suited for hardware acceleration, because operations are often repeated on every pixel and can be parallelized. Therefore, we chose to run a stereo vision computation on the discussed platform which is designed for a space robot used for a planetary surface exploration. The corresponding task graph is depicted in Fig. 4 as a Directed Acyclic Graph (DAG) with fork and join operations. First, both (left and right) camera raw images are converted from bayer to RGB format (*debayer*). In a second step, camera distortions are removed and both images are transformed into a common coordinate system (*rectify*). The *stereo match* task runs a Sobel filter over both images and computes a disparity map using the SAD block matching technique. This hardware task is a good example for different levels of parallelization: the number of disparities computed in parallel can be chosen freely at design time, which allows negotiating execution time with hardware resource usage. The *disparity to pointcloud* task generates a colored pointcloud from disparity map and the left rectified RGB image. Subsequently, the *pass through filter* task is applied in z direction (depth) of the pointcloud and cuts off implausible points. The extracted pointcloud is used as an input for different applications such as object recognition and visual odometry. We assume that the following applications require a minimum frame rate of $1\,\mathrm{fps}$, which is taken as the throughput criteria to satisfy.

All edges of the task graph are affected by transfer of streamable processing data. When executed as a hardware task, each edge requires either a DMA channel to be opened, or direct forwarding via the AXI Stream Interconnect. Some tasks additionally require parametrization data, e.g. the *rectify* task has to know if it operates on the left or right image.

The task graph is targeted to run on a platform such as the one depicted in Fig. 1 with $n = 2$ reconfigurable regions of different size and one processor core as Processing Elements (PEs). The smaller region is reconfigured in $8\,\mathrm{ms}$, the larger one in $18\,\mathrm{ms}$. Two DMA cores ($m = 2$) allow data transfer from memory to reconfigurable regions and vice versa.

Tasks requiring a complex hardware module with a large FPGA fabric footprint are restricted to the larger Region 2.

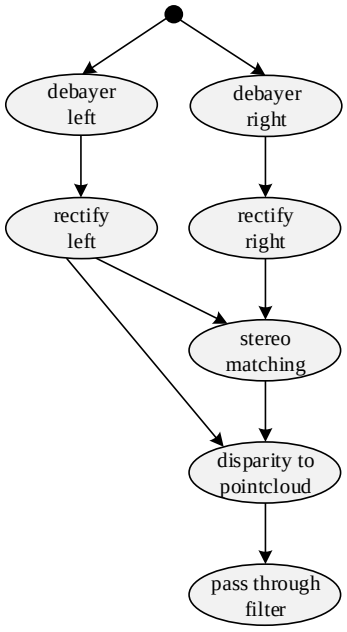

Fig. 4. Task graph for a stereo vision computation.

TABLE I
Execution Times for Stereo Vision Tasks [ms].

Task	Region 1 (small)	Region 2 (large)	Processor 1
reconfiguration / CSW	8	18	0
debayer	36	36	32
rectify	-	38	286
stereo match	456[1]	228	7469
disparity to pointcloud	528	528	615
pass through filter	-	-	412

[1] This value has not been measured in hardware execution yet, and has been estimated considering simulation results and hardware implementations with similar complexity.

This applies to the *rectify* task and the highly parallelized *stereo matching* (see Table I). Another *stereo matching* variant with lower parallelization and a smaller footprint also fits into Region 1. Simple computations (*debayer* and *disparity to pointcloud*) with low resource demands can be placed in both regions. All tasks are also implemented in software, using appropriate OpenCV functions. Up to now, the *pass through filter* has not yet been accelerated in hardware, hence this task may be executed in software only.

The execution times of all tasks in Table I have negligible jitter. This is because the presented image processing algorithms are not input dependent. Some tasks benefit more from hardware acceleration than others. E.g., the *rectify* task yields a speedup factor of 7.5 when executed in hardware. On the other hand, the *debayer* executes faster in software. This effect is caused by the higher processor core clock ($667\,\mathrm{MHz}$) compared to the slower hardware clock ($100\,\mathrm{MHz}$).

https://doi.org/10.24355/dbbs.084-202105041433-0

IV. Problem Description

The selected platform provides mechanisms for hardware acceleration of this task graph. However, the flexibility of outsourcing suitable computations to the programmable logic needs to be exploited appropriately. There exist numerous options for assigning the tasks of the (still quite simple) task graph of the introduced case study to PEs.

Many solutions already exist for task graph execution in software that also guarantee bounded latencies for real-time applications. This setup could be best compared to a multiprocessor problem statement as the platform provides multiple PEs for the execution of a task. However, due to the particular constraints and peculiarities that come with hardware acceleration using DPR, state-of-the-art multiprocessor scheduling strategies cannot be applied for the introduced platform. The following constraints differ from a pure multiprocessor scheduling problem:

- Hardware task execution is non-preemptive.
- Software task execution is not affected by the preceding constraint and furthermore preemptive.
- The reconfiguration process for switching between hardware tasks is many times longer than a context switch (CSW) in software.
- The reconfiguration process can be applied to only one region at a time.
- The amount of communication is limited. The number of concurrent DMA transfers is strictly upperbounded by a limited the number of read/write channels provided by the platform.
- Execution times of a task heavily depend on the selected PE. In general, the execution in hardware is faster compared to software, however some tasks do not benefit from hardware acceleration (see Table I).

Deciding when to move a computation from software as a hardware task to which reconfigurable region considering the given constraints is an optimization problem. Even for simple task graphs such as the one provided in the case study in Section III, this quickly grows to a sophisticated challenge.

Related work has already solved hardware (and software) task scheduling for various assumptions. [9] identified hardware task scheduling as an MILP problem and simplifies the calculation by scheduling a subset of the task graph at a time with k tasks per iteration. Based on those results, [10] formulated heuristics for an optimization of execution time and also takes the floorplanning of reconfigurable regions into account. [6] applies both Best Fit In Space and Best Fit In Time strategies and considers execution of tasks in both hardware and software. [11] virtualizes hardware tasks and treats reconfigurable regions as critical resources. A regions is locked for the duration of a hardware task. If a new hardware task tries to access a locked region, it has to stall until the lock is released and the contention is resolved.

Although almost all approaches optimize the execution time by using the optimization techniques *re-use* and *prefetching*, yet none exploit *parallelization* and *pipelining*. In general, communication bandwidth to/from reconfigurable regions is assumed to be unlimited, which conflicts with practical implementations. In order to analyze the communication appearance, task dependencies in a task graph need to be classified according to the type of data transfer. Huang et al. [12] prove that a scheduler reducing data movements between tasks also yields shorter processing times. However, their work does not target task-based reconfigurations of small FPGA areas.

In general multiple functions are mapped to one processing unit, resulting in a task graph that consists of connected components with different activation periods. This also has not yet been considered for hardware/software scheduling.

On this account, we developed an off-line solver for hardware/software scheduling on ARM-FPGA platforms that considers the named shortcomings. The main objective of the solver is to minimize latency and maximize throughput of a given task graph. Energy and power are handled as additional objectives.

A. Task Graph Characterization

Related work ([9], [13] and others) makes use of task graphs for restricting the task execution order. However, this is not sufficiently far-reaching. The platform needs to serve different communication needs and different types of data transfers between tasks, which needs to be considered for an efficient scheduling.

Tasks may exchange parameters only as depicted in Fig. 5(a), e.g., a threshold value returned by the calculation in task T1 is used as an input argument of task T2. In the given platform, such parameters can simply be passed via the AXI Lite Interconnect.

Other tasks exchange complete sets of processing data, which applies for many filters and arithmetic operations on regular data structures. This data needs to be transfered over the AXI Stream Interconnect. If the data is processed in the same sequential order, then it is *streamable* (Fig. 5(b)). If the processing order alters, it is not streamable (Fig. 5(c)) and has to be buffered in DDR memory.

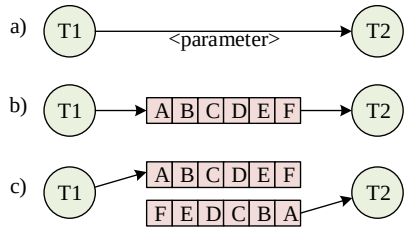

Fig. 5. Types of task dependencies: (a) parameter-only, (b) streamable, (c) non-streamable.

Many hardware tasks, such a repeated operation on every pixel of an image, execute the three steps *load*, *process*, and *store* in a pipelined manner for each datum. The overall measured execution time of these tasks cannot be split into time fractions for a load-, process-, and write phase. Other hardware tasks first load a complete dataset before processing it, which allows a more detailed timing analysis. However,

https://doi.org/10.24355/dbbs.084-202105041433-0

this does not affect the overall scheduling as the hardware module has to be configured in a region during all three phases. Therefore, it is sufficient to consider an overall execution time for each hardware task.

B. Latency and Throughput Objective

When minimizing the makespan (latency) and maximizing the repetition rate (throughput) of a task graph, multiple aspects have to be considered.

1) The *execution time* of a single task is assumed to be predictable, as input-independent algorithms produce very little jitter. However, the execution time depends on the selected PE. The performance of execution in software benefits from higher clock rates of the processor cores compared to the FPGA fabric. On the other hand, the lack of parallelization in a processor core usually results in execution times that are considerably longer. There are tasks that are more suitable for parallelization and yield higher speedup factors than others when executed in hardware. Additionally, the degree of parallelization correlates to the available resources, resulting in speedup factors dependent on the size of the reconfigurable region. Hence, it is not sufficient to minimize the idle time in each PE for a minimal makespan. There may be solutions with longer idle times but more efficient task-PE assignments and shorter execution times.

2) The need for *reconfiguration* depends on the task graph history. For an optimum schedule, the number of reconfiguration processes should be minimized by means of re-use. The reconfiguration time scales linearly with the reconfiguration area and is therefore accurately predictable for each region.

3) There are different reasons for *stall times* for tasks. These include waiting for a free PE, waiting for a free DMA channel, and blocking due to an already ongoing reconfiguration process.

The contributions of these three parts to the overall makespan depend on each other. Hence, an optimum solution needs to balance between shortest execution times, minimum number of reconfiguration processes, and minimal stall times.

C. Power and Energy Objective

Battery-powered devices always strive for reduced energy consumption. Devices with limited power supplies or unavailable heat sinks are also restricted in their peak power consumption. Both energy and power consumption can be estimated for memories, processors, and FPGA fabrics quite accurately, using device vendors' power models. Hardware task acceleration provides multiple options for the computation of a given task graph, which differ in energy and power efficiency. For example, a task executed in software generally consumes more power in conjunction with longer execution times. Because of the large variance of power and energy consumptions, the scheduling solver optimizes those parameters as an additional objective.

The calculation of the total energy and power consumed is split into several parts. For each PE, the static power $p_{static}(\mathrm{PE})$, consumed without executing any task, is specified. The dynamic power $p_{dyn}(\mathrm{PE}, \mathrm{task})$ specifies the additional consumption of a PE when a task is executed. Furthermore, the energy consumption for read and write memory transactions is computed.

TABLE II
POWER PROPORTIONS [mW].

Task	Region 1 (small)	Region 2 (large)	Processor 1
p_{static}	11	40	511
p_{dyn}(debayer)	6	6	276
p_{dyn}(rectify)	-	103	276
p_{dyn}(stereo match)	66[1]	132	279
p_{dyn}(disparity to pc)	16	18	297
p_{dyn}(pass through)	-	-	284

[1] This value has not been measured in hardware execution yet, and has been estimated considering simulation results and hardware implementations with similar complexity.

As an example, the separate power proportions have been calculated in Table II for the case study described in Section III. For memory transactions, $0.22\,\mathrm{mJ/Mbyte}$ have been calculated in read direction and $0.19\,\mathrm{mJ/Mbyte}$ in write direction. In the given case study, memory transactions contribute only a fraction to the total power consumption and will be neglected in the following calculations. While execution power varies little for different reconfigurable regions, the power demand for software execution is significantly higher.

V. ALGORITHM

In this section, we present an algorithm optimizing the energy consumption of the produced schedule subject to throughput and peak power usage constraints. If minimizing energy consumption is only a secondary goal, we can use our algorithm to optimize for throughput or latency as well. This can be done by performing binary search for the optimal throughput or latency value based on whether our algorithm finds a solution or reports the problem to be infeasible.

As input, our algorithm takes a description of all available regions and modules as well as a task graph. The task graph may consist of several connected components. Each component has to be processed periodically and has its own activation period; we assume harmonic periods, i.e., the longest occurring period (hyperperiod) T_{FPS} is a multiple of all periods. In order to deal with the periods, we copy each component according to the number of periods that fit into the hyperperiod. Each copy is assigned a release time corresponding to its period. We express throughput constraints as follows. Except for its first occurrence, each task graph component has a deadline corresponding to its period; this deadline is relative to the finish time of the first copy of the task graph. Optionally, an absolute deadline may be given as well. Moreover, the entire schedule must be repeatable after applying pipelining as depicted in Fig. 3. In other words, after pushing all tasks of iteration 1 of the schedule as far as possible to the left

https://doi.org/10.24355/dbbs.084-202105041433-0

(possibly overlapping iteration 0), all components in iteration 1 must also satisfy their deadlines; i.e., the only occurrence of a component that may take longer to execute than its period is the first occurrence of that component in iteration 0. A necessary condition for the schedule to satisfy the throughput constraints is that none of the processing elements is busy for longer than T_{FPS}. However, this condition is not sufficient due to DMA and power constraints.

The scheduling problem we consider generalizes several well-known NP-hard problems, such as PARTITION; therefore, we are dealing with a computationally hard problem. One of the most promising approaches to solving practical instances of computationally hard problems to optimality is the use of integer programming solvers such as IBM ILOG CPLEX [14] or GLPK [15]. Due to the large number of constraints, some of which are difficult to formulate efficiently in a linear fashion, directly solving a formulation of the entire scheduling problem as Mixed Integer Linear Program (MILP) is impractical. Moreover, we cannot expect to find an optimal solution for most larger instances in reasonable time; therefore, we need a search procedure that can be turned into an incomplete search that may miss the optimal solution, instead producing a solution of sufficient quality. In order to achieve this goal, we use a combination of combinatorial optimization methods and A^* search; there are several well-known variants of A^* that turn the complete search into a faster, incomplete one. In a first step, our algorithm generates the set $\mathcal{S}$ of *super tasks*; intuitively speaking, a supertask is a single task or a parallelized group of tasks together with a concrete, feasible mapping of the tasks to processing elements.

Definition 1. *A* super task *consists of a set of tasks S and a mapping ρ of these tasks either to processing elements, subject to the following constraints.*

- *For every task $t \in S$ with module m_t, it must be possible to execute m_t on processing element $\rho(t)$.*
- *The mapping must be injective, i.e., no processing element must contain more than one task.*
- *The tasks in S must all come from the same task graph T.*
- *The vertex-induced subgraph $T[S]$ must be (weakly) connected.*
- *All edges in $T[S]$ must be streamable.*
- *If ρ maps a task t to be executed in software, $S = \{t\}$.*
- *The number of data-afflicted edges coming into S in T must not be greater than the number of DMA channels. The same must hold for the edges going out of S.*
- *The sum of peak power requirements of all tasks must not exceed the limit.*

Potentially, the number of super tasks is exponential in the number of processing elements; however, the task graphs, DMA constraints and low number of regions typically keep the number of super tasks manageable so that we can explicitly enumerate the whole set $\mathcal{S}$.

Our search handles *states* consisting of initial pieces of a schedule. A state can be transformed into a successor state by applying one of the two following operations.

- Reconfiguring a region r to a new module m.
- Running a super task on a set of processing elements which must be properly configured.

Each state contains a pointer to its predecessor state together with the operation leading to this state. Moreover, it contains the *current time* t_{cur} corresponding to the beginning of the last operation, the current amount of power used, the energy cost so far and the current number of DMA channels used. It also keeps track of the time at which the last scheduled reconfiguration finishes. For each processing element g, a state contains information about the configuration of g and the last operation performed on g. For each module m, a state contains information about the number of tasks with module m that have not yet been scheduled. For each task s, a state contains information about whether the task was already scheduled. We measure time, energy consumption and power required using fixed-point arithmetic to avoid rounding issues.

When performing an operation, we have to determine the earliest point t_{next} in time where this operation may begin and update t_{cur} accordingly. We start the operation as soon as possible, disallowing any slack; this helps us to avoid handling each time individually. For reconfigurations, t_{next} is determined by the time at which the last scheduled reconfiguration is done and the reconfigured region stops executing. For execution of super tasks, this is determined by the time the last task and reconfigurations on the affected processing elements finish, by release times and by power usage and DMA channel constraints. Note that t_{next} need not be greater than t_{cur}; this can happen for instance if there is a processing element on which we have not scheduled an operation recently. If $t_{next} < t_{cur}$, we disallow performing the operation. If $t_{next} = t_{cur}$, we only allow the operation if it is greater than the previous operation w.r.t. some fixed order on the operations. This *time order rule* serves to break symmetries, i.e., to prevent considering identical states multiple times. Furthermore, we disallow any reconfiguration on a region for which the last operation has been a reconfiguration. Additionally, we discard any state with violated deadlines. Moreover, we disallow any reconfiguration of a region to a module that does not have to be executed anymore according to the counter stored in the state. These rules help to avoid obviously suboptimal solutions and ensure that the longest solution we have to consider has $2n$ operations, where n is the number of tasks; it consists of one reconfiguration and one single-task execution for each task.

In order for A^* to perform reasonably well, the A^* algorithm requires a good heuristic to estimate the cost for transforming a state corresponding to a partial solution into a goal state corresponding to a complete solution. The algorithm is only complete if the heuristic *underestimates* this cost; in our case, this means that we need an estimate for the energy cost. In order to reduce the number of states we have to consider, we also use a heuristic that checks whether our throughput constraints can still be met; if this is not the case for a certain state, we can simply discard that state. In

https://doi.org/10.24355/dbbs.084-202105041433-0

the following, we describe an integer linear program H that can simultaneously check for the feasibility of the throughput constraints and produce a lower bound on the remaining energy consumption. H relaxes several aspects of the problem (such as DMA and power constraints and task dependencies) and is thus not suitable to solve the problem exactly. Solving an integer program for each state is too expensive; therefore, we only solve the LP relaxation of the problem. However, integer programming solvers such as CPLEX include several families of cutting planes that can be used to strengthen the LP relaxation. Letting the solver generate such cutting planes helps improving the accuracy of the heuristic at the expense of more computation time per state.

For each region g and each module m, the integer program H contains a variable $r_{g,m} \in \{0,1\}$ that indicates whether g must be reconfigured to m. For each task s that is not yet scheduled and each processing element g that s can be executed on, H contains a variable $t_{s,g} \in \{0,1\}$ that indicates whether s is executed on processing element g. For each processing element g, let T_g be the maximum of t_{cur} and the time until which g is busy. Moreover, let $T(s,g)$ be the execution time of s on g, let $P(s,g)$ be the power consumption while executing s on g and $T_{rec}(g)$ be the reconfiguration time of g. For each processing element g, let I_g be the idle time on g at the beginning of our solution; if g is empty, let $I_g = T_g$. For each region g, we can express the remaining execution time on g as

$$T_{rem}(g) := \sum_{\text{Task } s} T(s,g)t_{s,g} + \sum_{\text{Module } m} T_{rec}(g)r_{g,m};$$

for software execution, the right summand corresponding to reconfiguration times can be ignored. Additionally, let H include a variable $T_{\text{MAX}} \in \mathbb{N}$ corresponding to the maximum busy time among all processing elements. We can express the energy $P_{rem}(g)$ used on the remainder of the execution on a processing element g as

$$(T_{\text{MAX}} - T_g + I_g) \cdot p_{static}(g) + \sum_{\text{Task } s} P(s,g)T(s,g)t_{s,g}.$$

The integer program H is defined as follows.

$$\min \quad \sum_{\text{Region } g} P_{rem}(g) \qquad \text{s.t.}$$

$$\forall s: \quad \sum_{\text{Region } g} t_{s,g} = 1 \tag{1}$$

$$\forall g \forall s \text{ with module } m: \quad r_{g,m} \geq t_{s,g} \tag{2}$$

$$\forall g: \quad T_g + T_{rem}(g) - I_g \leq T_{\text{MAX}} \tag{3}$$

$$T_{\text{MAX}} \leq T_{\text{FPS}} \tag{4}$$

If H is infeasible for a state, this state can be discarded; it is not possible to satisfy the throughput constraints starting from this state. On the other hand, if H is feasible, its solution value can be used to compute a heuristic that underestimates the total energy cost of the state.

TABLE III
MAKESPAN AND THROUGHPUT IMPROVEMENT PER OPTIMIZATION TECHNIQUE.

Optimization Technique	Makespan Reduction	Throughput Gain	Energy Savings
Re-use	2 %	3 %	2 %
Prefetching	0.6 %	3 %	2 %
Parallelization	3 %	4 %	3 %
Pipelining	-	50 %	26 %
Relocation	0 %	0 %	0 %

VI. EVALUATION

Once fed with the task execution times from Table I and platform restrictions ($m = 2$, $n = 2$), the scheduling algorithm outputs the Gantt chart depicted in Fig. 6 as an optimum schedule. Both *debayer* and *rectify* modules are re-used when executing the corresponding task on left and right image data. Parallelization is applied between *debayer* and *rectify* tasks. The total makespan of the task graph is $1288\,\text{ms}$. By pipelining two iterations of the task graph, a repetition rate of up to $1.166\,\text{fps}$ (a period of $858\,\text{ms}$) can be achieved, satisfying the throughput requirement using $646.032\,\text{mJ}$ per iteration including static power usage.

For the stereo vision case study, we additionally calculated the makespan and throughput with deactivated optimization techniques identified in Section II. Table III lists the achieved performance gains and energy savings for each optimization technique.

Both re-use and prefetching do not improve the overall solution significantly. The contribution of reconfiguration time to the total makespan is quite small, hence these two techniques saving reconfiguration time have little effect.

Making use of parallelization produces only a small performance increase, because it is applied to the tasks with shortest execution times (*debayer* and *rectify*). However, this feature can show its strength when applied to tasks with longer execution times.

For the presented case study, pipelining yields a throughput gain of $50\,\%$. It exploits that the next iteration of the task graph may start while the last task (*pass through filter*) is still being executed on Processor 1. We have to state, that pipelining yields an exceptional large gain for the given case study. We expect a large mean variation for throughput gains through pipelining depending on the scheduling problem. Note that pipelining does not reduce the makespan of a single iteration of the task graph. Only when considering multiple iterations, this technique increases the repetition rate and throughput.

The relocation feature is not supported by our platform. Still, we tried to further optimize the schedule using this technique. However, no improved solution has been found. This can be explained by the added reconfiguration overhead that worsens the overall schedule when interrupting a hardware task.

https://doi.org/10.24355/dbbs.084-202105041433-0

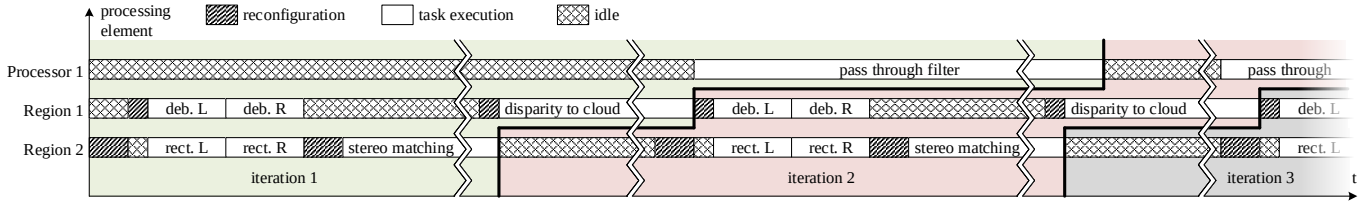

Fig. 6. Gantt chart for the stereo vision case study.

The ability of the scheduling algorithm to process multiple connected components in a task graph with different activation periods has not been exploited in the given case study. However, it is an important feature for general hardware/software scheduling problems. As a next step in our case study, object recognition and visual odometry is being implemented with hardware acceleration, both using the already generated point-cloud. Generally, the visual odometry is a simpler computation compared to object recognition, however it requires a higher frame rate and hence a different activation period. With the developed scheduling algorithm, such problem statements can be solved.

Furthermore, it allows to consider interfering processing load. A separate task with a distinct activation period can conservatively reserve execution time on processor cores for background tasks and interrupt handlers. This is a good method for modeling additional processing load not related to the actual data-intensive computation. For processor cores, such interference is evident, e.g., interrupt handlers with a higher priority may interrupt the actual computation of a task graph. But also hardware tasks can be affected by interference. Our stereo vision case study is part of an autonomous space robot used for exploration. When operating in such environments, SRAM-based FPGAs are exposed to radiation that induces Single Event Upsets (SEUs). Once detected, those errors require countermeasures, e.g. re-running the last computation step. Such repetition events cannot be planned in advance, however our scheduling algorithm allows to model interference and conservatively reserve execution time on reconfigurable regions.

VII. Conclusion

The overall performance of an embedded system can be increased by outsourcing the execution of specific tasks into the programmable logic of an ARM-FPGA platform using Dynamic Partial Reconfiguration. However, long reconfiguration times, a limited number of DMA channels, and further constraints restrict the scheduling of task graphs using this feature. The developed algorithm optimizes both makespan and throughput. Additionally, the power and energy consumption can be minimized. We showed that making use of different optimization techniques improves the solution appreciably. Not only existing strategies (re-use and prefetching), but also new strategies developed by us (parallelization and pipelining) increase both makespan/throughput and power/energy efficiency for our stereo vision case study.

Acknowledgment

This work is part of the DFG Research Group FOR 1800 ”Controlling Concurrent Change”. Funding for the IDA was provided under grant number MI 1172/3-1; for the Algorithms Group FE 407/17-2.

References

[1] Genode Labs GmbH, “Genode operating system framework,” https://genode.org/, accessed: 2018-02-28.

[2] A. Dörflinger, M. Albers, B. Fiethe, and H. Michalik, “Hardware acceleration in Genode OS using dynamic partial reconfiguration,” in *Proceedings of the 31st Int. Conference on Architecture for Computing Systems (ARCS 2018)*, ser. LNCS, in press, 2018, pp. 306–318.

[3] *Zynq-7000 All Programmable SoC TRM, UG585*, v1.12.1 ed., Xilinx Inc., 2017.

[4] A. Lomuscio, G. C. Cardarilli, A. Nannarelli, and M. Re, “A hardware framework for on-chip FPGA acceleration,” in *International Symposium on Integrated Circuits (ISIC)*, Dec 2016, pp. 1–4.

[5] M. Pagani, M. Marinoni, A. Biondi, A. Balsini, and G. Buttazzo, “Towards real-time operating systems for heterogeneous reconfigurable platforms,” in *12th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, July 2016, pp. 49–54.

[6] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, “Hardware task scheduling for partially reconfigurable FPGAs,” in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2015, pp. 487–498.

[7] T. Xia, J.-C. Prévotet, and F. Nouvel, “Microkernel dedicated for dynamic partial reconfiguration on ARM-FPGA platform,” *SIGBED Rev.*, vol. 11, no. 4, pp. 31–36, Jan 2015.

[8] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, “Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs,” in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE ’16. San Jose, CA, USA: EDA Consortium, 2016, pp. 1505–1508.

[9] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio, “A multiobjective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures,” in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, pp. 1–6.

[10] A. Purgato, D. Tantillo, M. Rabozzi, D. Sciuto, and M. D. Santambrogio, “Resource-efficient scheduling for partially-reconfigurable FPGA-based systems,” in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 189–197.

[11] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell, “Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform,” *Journal of Signal Processing Systems*, vol. 77, no. 1, pp. 61–76, Oct 2014.

[12] M. Huang, H. Simmler, P. Saha, and T. El-Ghazawi, “Hardware task scheduling optimizations for reconfigurable computing,” in *Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, Nov 2008, pp. 1–10.

[13] L. Pezzarossa, M. Schoeberl, and J. Sparsø, “Reconfiguration in FPGA-based multi-core platforms for hard real-time applications,” in *11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, June 2016, pp. 1–8.

[14] “IBM ILOG CPLEX Optimizer,” https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer, accessed: 2018-03-16.

[15] “GNU Linear Programming Kit,” https://www.gnu.org/software/glpk, accessed: 2018-03-16.

https://doi.org/10.24355/dbbs.084-202105041433-0